

华中科技大学

课程实验报告

课程名称： 汇编语言程序设计实践

专业班级： 计算机科学与技术 2006 班

学 号： U202015471

姓 名： 杨释钧

指导教师： 张勇

实验时段： 2022 年 3 月 7 日~6 月 6 日

实验地点： 东九 A314

原创性声明

本人郑重声明：本报告的内容由本人独立完成，有关观点、方法、数据和文献等的引用已经在文中指出。除文中已经注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品或成果，不存在剽窃、抄袭行为。

特此声明！

学生签名：

报告日期：2022. 6. 6

实验报告成绩评定：

一（50 分）	二（35 分）	三（15 分）	合计（100 分）

指导教师签字：

日期：

目录

一、程序设计的全程实践	1
1.1 目的与要求	1
1.2 实验内容	1
1.3 内容 1.1 的实验过程	1
1.3.1 设计思想	1
1.3.2 流程图	2
1.3.3 源程序	4
1.3.4 实验记录与分析	11
1.4 内容 1.2 的实验过程	14
1.4.1 实验方法说明	14
1.4.2 实验记录与分析	14
1.5 小结	17
1.5.1 主要认识与收获	17
1.5.2 思考题	17
1.5.3 实验操作的经验教训	18
二、利用汇编语言特点的实验	19
2.1 目的与要求	19
2.2 实验内容	19
2.3 实验过程	19
2.3.1 实验方法说明	19
2.3.2 实验记录与分析	20
2.4 小结	26
2.4.1 认识与收获	26
2.4.2 思考题	27
2.4.3 经验教训	27
三、工具环境的体验	28
3.1 目的与要求	28
3.2 实验过程	28
3.2.1 WINDOWS10 下 VS2019 等工具包	28
3.2.2 DOSBOX 下的工具包	31
3.2.3 QEMU 下 ARMv8 的工具包	31
3.3 小结	34

汇 编 语 言 程 序 设 计 实 验 报 告

3.3.1 VS2019.....	34
3.3.2 DosBox.....	34
3.3.3 QEMU 下的 ARMv8 的工具包	34
参考文献	35

一、程序设计的全程实践

1.1 目的与要求

1. 掌握汇编语言程序设计的全周期、全流程的基本方法与技术；
2. 通过程序调试、数据记录和分析，了解影响设计目标和技术方案的多种因素。

1.2 实验内容

内容 1.1：采用子程序、宏指令、多模块等编程技术设计实现一个较为完整的计算机系统运行状态的监测系统，给出完整的建模描述、方案设计、结果记录与分析。

内容 1.2：初步探索影响设计目标和技术方案的多种因素，主要从指令优化对程序性能的影响，不同的约束条件对程序设计的影响，不同算法的选择对程序与程序结构的影响，不同程序结构对程序设计的影响，不同编程环境的影响等方面进行实践。

1.3 内容 1.1 的实验过程

1.3.1 设计思想

- 算法思想：主要使用简单模拟的思路，将任务书中的相关操作直接实现。
- 模块划分与说明：为降低整个系统代码的耦合性，在本次任务中，整个系统的实现被划分成两个部分，相关代码分别放在 `main.asm` 文件和 `func.asm` 文件中。`Func.asm` 文件存放了实现本次任务所实现的系统中相关的功能的函数及其辅助函数，`main.asm` 文件中只实现了 `main` 函数，主要负责提供用户输入时打印提示信息以及调用 `func.asm` 文件中实现的函数来实现系统要求功能。
- 模块间公共符号说明、段是否合并说明：
模块间公共符号包括打印输出固定格式以及一些各个模块间公共的全局变量：

```
public lpFmt
```

```
public lpFmt3
```

```
public MIDF
```

```
public f
```

```
public y
```

公用函数为：

```
Scopy proto:dword, :dword
```

```
func proto:dword
```

```
MIDprint proto
```

- 相关函数信息：下面将分文件介绍相关函数的作用。

Func.asm：

(1) `func proc data:dword`

功能：该函数接收一个 `dword` 型变量，在系统中主要负责根据当前这组信息计算对应的 `f` 值，其中参数 `data` 存放当前需要计算的信息。

汇编语言程序设计实验报告

(2) Scopy proc des:dword,sou:dword

功能：该函数接收两个 dword 型变量，将变量 sou 的值拷贝到变量 des，在系统中主要负责根据 func 函数计算得到的 f 值来讲当前这组信息拷贝到相应的存储区。

(3) MIDprint proc

功能：该函数在系统中主要负责实现功能 3，即将目前存储在 MIDF 区域的数据打印出来。

Main.asm:

(1) strcmp macro str1, str2, flag

功能：该宏主要实现字符串比较功能，共接受三个参数，其中 str1 和 str2 为待比较字符串，flag 用来作为返回值，供系统判断这两个字符串是否相同

(2) main proc c

功能：作为整个系统的入口，负责打印对用户的提示信息，调用 func.asm 模块中的相关辅助函数来实现整个系统所需要的功能。

● 寄存器分配：

Eax：在宏 strcmp 中作为拷贝的索引计数器，在调用函数 func 计算 f 的值时作为计算次数的计数器。

Ebx：在函数 Midprint 中作为计数器，指示当前打印存储区的位置，在函数 Scopy 中作为计数器，指示当前拷贝的位置。

Esi：在主程序、func、Midprint、Scopy 中均作为临时存储的变量。

Edi：在函数 Scopy 中储存源地址。

1.3.2 流程图

主程序流程图如图 1.1 所示。

汇编语言程序设计实验报告

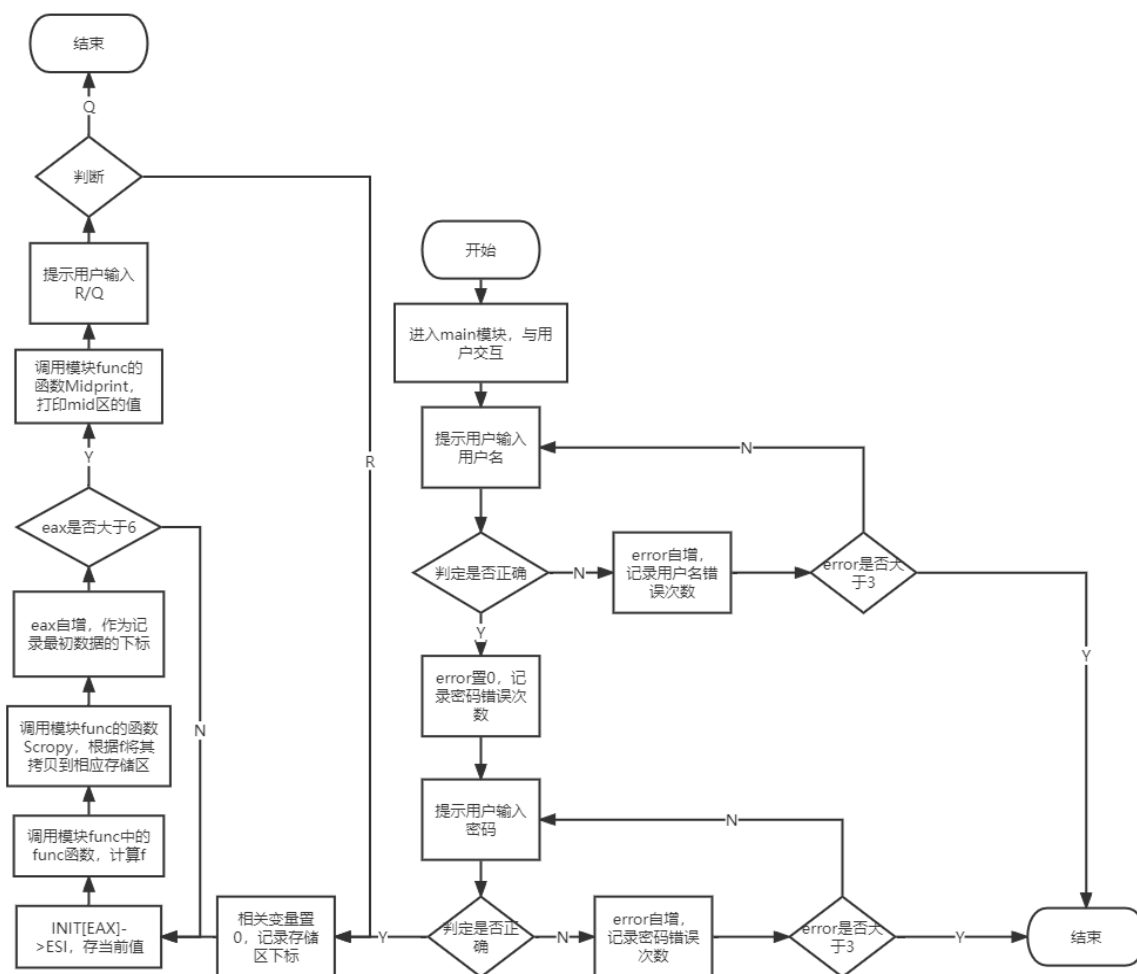


图 1.1 主程序流程图

func 函数流程图如图 1.2 所示。

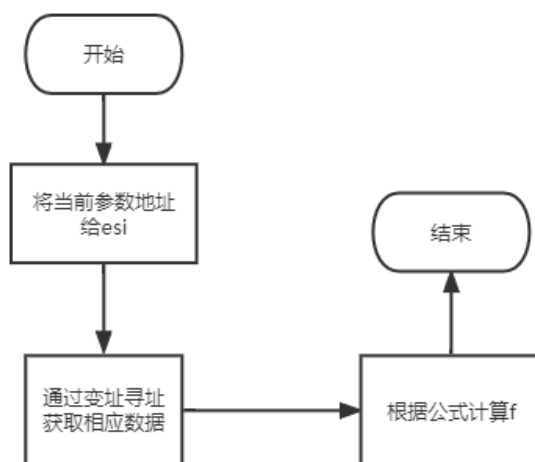


图 1.2 func 函数流程图

汇编语言程序设计实验报告

Midprint 函数流程图如图 1.3 所示。

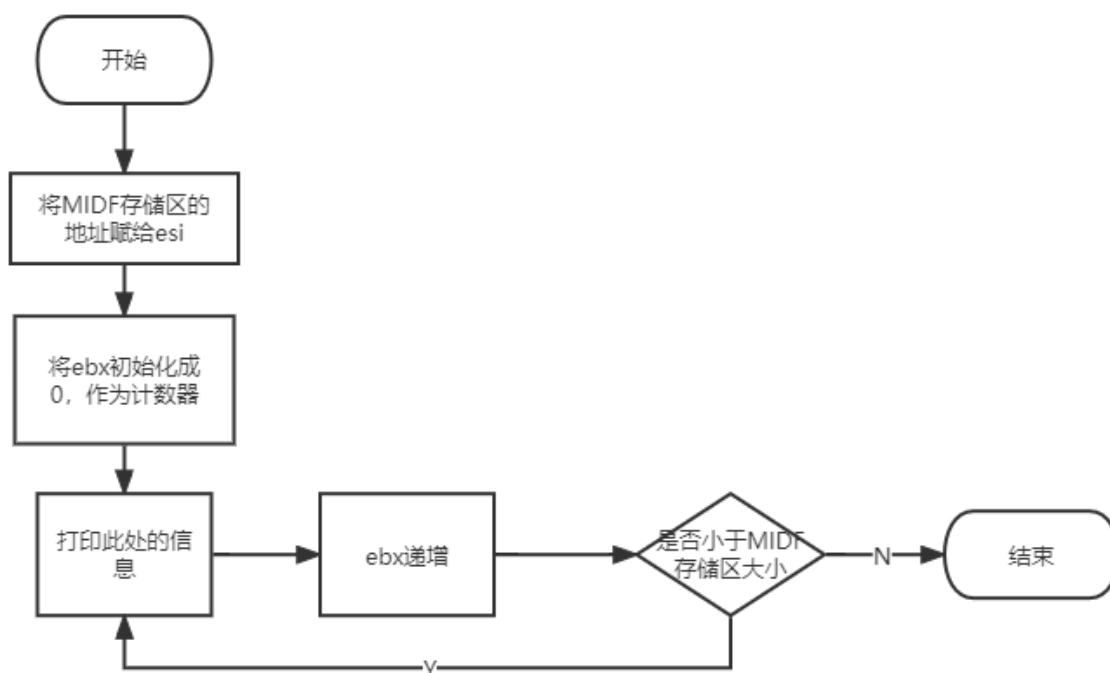
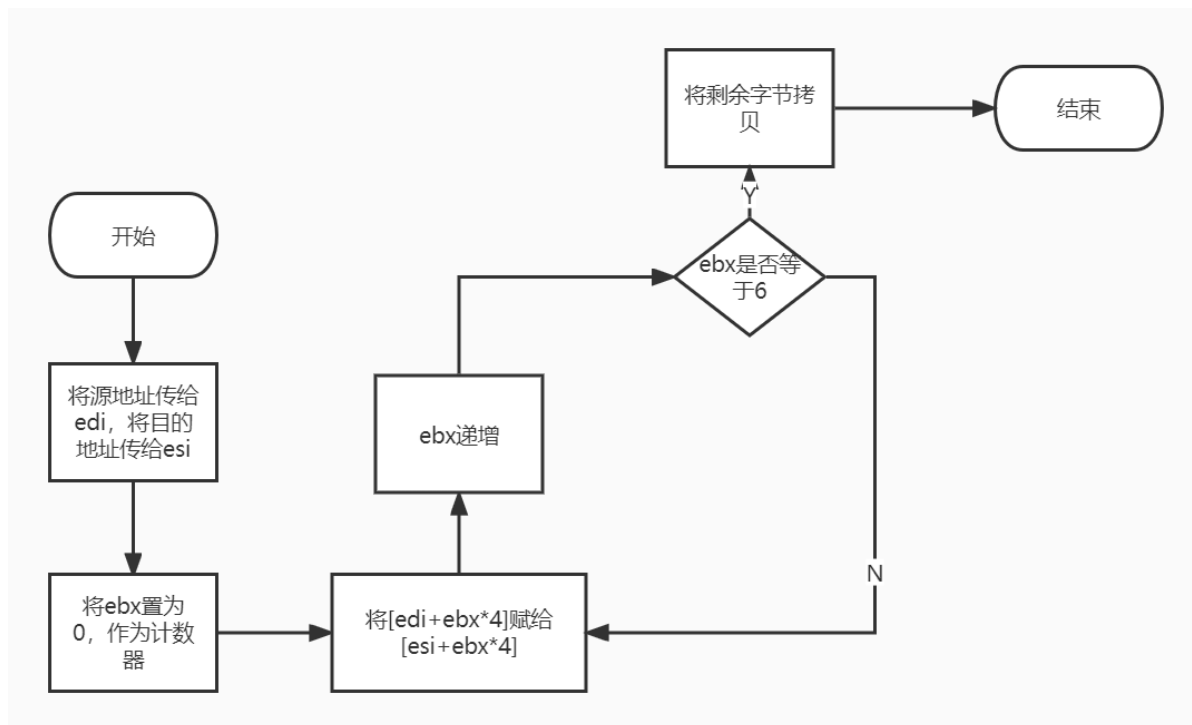


图 1.3 Midprint 函数流程图

Scopy 函数流程图如图 1.4 所示。



1.3.3 源程序

;main.asm 模块代码如下

.686

汇编语言程序设计实验报告

```
.model flat, stdcall
ExitProcess PROTO STDCALL :DWORD
includelib kernel32.lib
printf PROTO C :VARARG
scanf PROTO C :vararg
Scopy proto:dword, :dword
func proto:dword
MIDprint proto
includelib libcmnt.lib
includelib legacy_stdio_definitions.lib
public lpFmt
public lpFmt3
public MIDF
public f
public y
strcmp macro str1, str2, flag
    local S0, S1, S2
    push eax
    push ebx
    mov bh, 0
    mov eax, 0
S0:
    mov bh, str1[eax]
    cmp bh, str2[eax]
    jne S1
    inc eax
    cmp bh, 0
    jne S0
    mov flag, 1
    jmp S2
S1:
    mov flag, 0
S2:
    pop ebx
    pop eax
endm
S struct
    SAMID DB 9 dup(0)
    SDA DD 0
    SDB DD 0
    SDC DD 0
```


汇编语言程序设计实验报告

```
SF DD 0
S ENDS
. DATA
    lpFmt db "%s", 0ah, 0dh, 0
    lpFmt1 db "%s", 0
    lpFmt2 db "%s:%s", 0ah, 0dh, 0
    lpFmt3 db "%d", 0ah, 0dh, 0
    a dd 0
    x dd 0
    y dd 0
    z dd 0
    f dd 0
    de dd 0
    N dd 6
    so dd 0
    LOWF S 5 dup(<>)
    MIDF S 5 dup(<>)
    HIGHF S 5 dup(<>)
    INIT S
    <' 00000000', 2539, 6, 1, >, <' 00000001', 2539, 6, 1, >, <' 00000002', 2540, 1, 1, >, <' 00000003', 2540
    , 1000, 1, >,
        <' 00000004', 2540, 44, 44, >, <' 00000005', 2540, 666, 666, >, <' 00000006', 5, 4, 1, >
    STR1 db "please enter your username", 0
    STR2 db "please enter your password", 0
    str3 db "Rerun or Quit(R/Q)", 0
    password db '123456', 0
    username db 'ysj', 0
    output1 db 'OK!', 0
    output2 db 'Incorrect Input!', 0
    flag db 0
    error db 0
    opt db ?
    input db 7 dup(0)
. STACK 200
. CODE
main proc c
L0:
    invoke printf, offset lpFmt, OFFSET STR1
    invoke scanf, offset lpFmt1, offset input
    strcmp username, input, flag
    cmp flag, 1
```

汇编语言程序设计实验报告

```
jne L1
mov error, 0
invoke printf, offset lpFmt, OFFSET STR2
invoke scanf, offset lpFmt1, offset input
strcmp password, input, flag
cmp flag, 1
jne L1
jmp L2
L1:
    invoke printf, offset lpFmt, OFFSET output2
    inc error
    cmp error, 3
    je exit
    jmp L0
L2:
    mov a, 0
    mov x, 0
    mov y, 0
    mov z, 0
    mov f, 0
L3:
    cmp a, 6
    je L7
    mov eax, a
    IMUL EAX, TYPE S
    lea esi, INIT[eax]
    mov so, esi
    mov f, 0
    invoke func, so
    INC a
    cmp f, 100
    jl L4
    je L5
    jg L6
L4:
    mov eax, x
    imul eax, TYPE S
    inc x
    lea esi, LOWF[eax]
    mov de, esi
```

汇编语言程序设计实验报告

```
    invoke Scopy, de, so
    jmp L3
L5:
    mov eax, y
    imul eax, TYPE S
    inc y
    lea esi, MIDF[eax]
    mov de, esi
    invoke Scopy, de, so
    jmp L3
L6:
    mov eax, z
    imul eax, TYPE S
    inc z
    lea esi, HIGHF[eax]
    mov de, esi
    invoke Scopy, de, so
    jmp L3
L7:
    call MIDprint
    invoke printf, offset lpFmt, offset str3
    invoke scanf, offset lpFmt2, offset opt
    mov al, opt
    cmp al, 82
    je L2
    cmp al, 114
    je L2
exit:
    invoke ExitProcess, 0
main endp
END
```

;func. asm 模块代码如下:

```
. 686
.model flat, stdcall
ExitProcess PROTO STDCALL :DWORD
includelib kernel32.lib ; ExitProcess 在 kernel32.lib中实现
printf      PROTO C :VARARG
scanf       PROTO C :vararg
includelib libcmt.lib
```

汇编语言程序设计实验报告

```
includelib legacy_stdio_definitions.lib
extern lpFmt:byte
extern lpFmt3:byte
extern f:dword
extern y:dword
S struct
    SAMID DB 9 dup(0)
    SDA    DD 0
    SDB    DD 0
    SDC    DD 0
    SF     DD 0
S ENDS
extern MIDF:S
.data

.stack 200
.code
Scopy proc des:dword, sou:dword
    push esi
    push edi
    push eax
    push ebx
    mov ebx, 0
    mov esi, sou
    mov edi, des
S0:
    mov eax, [esi+ebx*4]
    mov [edi+ebx*4], eax
    inc ebx
    cmp ebx, 6
    jne S0
    mov al, [esi+24]
    mov [edi+24], al
    pop ebx
    pop eax
    pop edi
    pop esi
    ret
Scopy endp
func proc data:dword
    push esi
```

汇编语言程序设计实验报告

```
    push ebx
    mov esi, data
    mov ebx, [esi+9]
    imul ebx, 5
    sub ebx, [esi+17]
    add ebx, [esi+13]
    add ebx, 100
    shr ebx, 7
    mov [esi+21], ebx
    mov f, ebx
    pop ebx
    pop esi
    ret
func endp
MIDprint proc
    push esi
    push ebx
    push eax
    mov ebx, 0
    lea esi, MIDF
L0:
    invoke printf, offset lpFmt, esi
    invoke printf, offset lpFmt3, MIDF[ebx].SDA
    invoke printf, offset lpFmt3, MIDF[ebx].SDB
    invoke printf, offset lpFmt3, MIDF[ebx].SDC
    invoke printf, offset lpFmt3, MIDF[ebx].SF
    add ebx, TYPE S
    add esi, TYPE S
    mov eax, y
    imul eax, TYPE S
    cmp ebx, eax
    jne L0
    pop eax
    pop ebx
    pop esi
    ret
MIDprint endp
end
```

汇编语言程序设计实验报告

1.3.4 实验记录与分析

1. 实验环境条件

硬件环境:

- 处理器: AMD Ryzen 5 4600H with Radeon Graphics 3.00 GHz
- 机带 RAM: 16.0GB(15.4GB 可用)
- 系统类型: 64 位操作系统, 基于 x64 的处理器

软件环境:

- IDE: WINDOWS10 下 VS2019 社区版

2. 汇编、链接中的情况

汇编过程中出现了 2 个问题,第一个问题是关于汇编链接的问题,一直报错说找不到相关函数,后来发现是主程序调用其他模块的子程序的时候忘记使用 `extern` 声明了。

第二个问题是关于头文件说明的问题,最开始不同模块有的是 `.model flat, stdcall`,有的是 `.model flat, c`,但是进行编译的时候程序会报错,后来经过排查才找到了这个问题。

3. 程序基本功能的验证情况

(1) 对功能一的测试。

进入系统时提示用户输入用户名,运行结果如图 1.5 所示。

```
please input your username
```

图 1.5 进入系统界面

先测试输入正确时系统的反应,输入的用户名与密码均正确时,按照系统设定,并不会给出相关的提示语句,而是继续进行系统的相关操作。运行结果如图 1.6 所示。

```
please input your username
ysj
please input your password
123456
00000000
2539
6
1
100
00000001
2539
6
1
100
00000002
2540
1
1
100
00000004
2540
44
44
100
00000005
2540
666
666
100
Rerun or Quit(R/Q)
```

图 1.6 当密码与用户名均正确时的运行结果

接下来测试输入不正确的情况,首先当连续输入三次错误的用户名时,程序在每次读入错误用户名时都会给出提示,在连续得到三次错误的用户名时会直接退出程序,运行结果如图 1.7 所示。

汇编语言程序设计实验报告

```
please input your username
dsa
Incorrect Input!
please input your username
dsafc
Incorrect Input!
please input your username
dwawd
Incorrect Input!
D:\vsProject\lab3\Debug\lab3.exe (进程 9260)已退出, 代码为 0。
按任意键关闭此窗口. . .
```

图 1.7 连续三次输错用户名时的运行结果

当输入的用户名正确, 但是连续输错三次密码时, 程序在每次读入错误密码时都会给出提示, 在连续得到三次错误的密码时会直接退出程序, 运行结果如图 1.8 所示。

```
please input your username
dsafd
Incorrect Input!
please input your username
fsahi
Incorrect Input!
please input your username
ysj
please input your password
121
Incorrect Input!
please input your username
3214
Incorrect Input!
please input your username
213421
Incorrect Input!
D:\vsProject\lab3\Debug\lab3.exe (进程 4292)已退出, 代码为 0。
按任意键关闭此窗口. . .
```

图 1.8 用户名正确但是连续三次输入密码的运行结果

(2) 对功能 3 的测试

功能三需要打印此时 MIDE 存储区的内容, 根据前文提到的算法, 该功能应该打印出的内容应该是<'00000000', 2539, 6, 1, >, <'00000001', 2539, 6, 1, >, <'00000002', 2540, 1, 1, >, <'00000004', 2540, 44, 44, >, <'00000005', 2540, 666, 666, >的值。实际运行结果如图 1.9 所示。

```
ysj
please input your password
123456
00000000
2539
6
1
100
00000001
2539
6
1
100
00000002
2540
1
1
100
00000004
2540
44
44
100
00000005
2540
666
666
100
Rerun or Quit(R/Q)
```

图 1.9 打印存储区内容运行结果

可以看出运行结果是正确的。

(3) 对功能 4 进行测试

功能四需要用户选择是重复执行还是退出, 当用户选择重复执行时, 运行结果如图 1.10 所示。

汇编语言程序设计实验报告

```
Rerun or Quit(R/Q)
R
00000000
2539
6
1
100
00000001
2539
6
1
100
00000002
2540
1
1
100
00000004
2540
44
44
100
00000005
2540
666
666
100
Rerun or Quit(R/Q)
```

图 1.10 对功能 4 重复执行选项的测试

当用户选择退出时，运行结果如图 1.11 所示。

```
Rerun or Quit(R/Q)
Q
D:\vsProject\lab3\Debug\lab3.exe (进程 8556) 已退出，代码为 0。
按任意键关闭此窗口。...
```

图 1.11 对功能 4 退出选项的测试

4. 使用调试工具观察、探究代码的情况

其实这一部分在编写的时候还是比较顺利的，没有出现太多让人印象很深刻的问题，我在这里主要想说一个在实验 2 中遇到的问题，正是因为实验 2 中遇到的这个问题让我有意识地在实验三中调用函数时去进行保护现场。

在实验二中，最后计时完毕后，由于 GetTickCount 函数将返回值放到 eax 寄存器中，所以直接用该寄存器中的值减去开始时间就可以得到总运行时间，但是在实际输出的过程中，尽管总运行时间很长，但是输出的 eax 值却仅仅只有 18，这一部分的代码如图 1.12 所示。

```
invoke printf,offset IpFmt2,offset str2
;pop eax
invoke printf,offset IpFmt,eax
```

图 1.12 打印运行时间代码

其中第一个 printf 函数是用来打印提示语句的。

后来我利用 vs2019 单步调试，在运行到第一个 printf 函数时，eax 的值如图 1.13 所示。

EAX = 000084FF

图 1.13 eax 的值 1

EAX = 00000012

图 1.14 eax 的值 2

很明显这个值是比较正常的，然后当我跳过这个 printf 函数之后，eax 的值如图 1.14 所示。

可以看出在 printf 函数中 eax 寄存器的值发生了改变，所以我们需要在调用之前进行保护现场的操作。

经过这个错误之后，在实验三中我开始有意识地保护现场，所以并没有出现因为这个错误导致的问题。

5. 其他

在任务 3.1 中，我研究了主程序和子程序之间传递参数的一些方法，比如可以通过栈来传递参

汇编语言程序设计实验报告

数，也可以通过约定寄存器来传递参数。

1.4 内容 1.2 的实验过程

1.4.1 实验方法说明

1. 指令优化对程序的影响

依托于实验二，这里采用系统函数 `GetTickCount` 函数进行计时，仅仅记录计算过程消耗的时间，不把打印结果的 `printf` 函数运行时间计算在内，此外，为了保证运行时间的稳定性，本次实验将计算过程重复十亿次，这里是内循环十次，外循环一亿次，最终运行时间较为稳定。

另外经过尝试我发现如果在数据段定义很大的存储区的话，程序编译速度会变得非常慢。例如在本次实验中，用 `INIT` 数组来存储最开始的结构体数据，如果将其定义成 `INIT S 100000000 dup(<, 256809, -1023, 1265, >)`，整个编译速度非常的长，所以最终还是考虑将 `INIT` 数组长度设置成 10，外循环一亿次来计算运行时间。我想出现这种问题可能是要进行大量的内存读写操作导致的。

共尝试了 3 种指令优化方法，有的效果还不错，有的效果提升并不明显，三种方法分别为：

- 乘除改为移位操作。本以为将乘除都改成移位操作会有比较大的提升，但是实际上提升并没有自己想象的那么大，一方面是本实验中仅仅涉及到了一个乘法和一个除法，而乘法仅仅是乘 5，将其改成左移 2 位加其本身两条指令可能并不会有很大提升，而将除以 128 改成右移 7 位还是有一点提升的。
- 拷贝时从逐字节拷贝改为除了流水号逐字节拷贝，其他的数据按每四个字节拷贝。这里的优化思路类似于循环展开，减少了循环次数，同时减少了整个过程中的关键路径的长度，从而得到更大的性能提升。
- 将流水号的拷贝方式从逐字节拷贝更改成两次四字节拷贝和逐字节拷贝剩余字符。优化思路与其合理性说明类似于第二点，此处不再赘述。

2. 约束条件、算法与程序结构的影响

本次实验中所要求实现的主要程序功能类似，其中决定程序性能的主要要求就是内容拷贝，在实验 2 中我们主要进行优化的也就是内容拷贝的部分，而在后续实验 5 中在 `QEMU` 下 `ARMv8` 工具包中也进行了对于内容拷贝优化的探讨。通过这两部分，我们可以看到通过从逐字节拷贝依次更换为逐字拷贝、逐多字拷贝，从算法的方面进行循环展开，尽可能减小关键路径的长度，从而使程序能够具有更好的性能。

3. 编程环境的影响

本次实验先后于 `Windows` 环境下在 `vs2019` 中调试运行了 32 位程序和 64 位程序，探究 32 位程序和 64 位程序的异同点。

后续，也在 `QEMU` 下 `ARMv8` 的工具包下编译了汇编程序，体会了该体系与 `80X86` 体系的异同，主要关注了 `CPU` 内寄存器、段的定义方法、指令语句及格式的特点、子程序调用的参数传递与返回方法、与 `C` 语言混合编程等特点。

1.4.2 实验记录与分析

1. 优化实验的效果记录与分析

优化一：乘除改为移位操作。（优化 1/10000000ms-3/10000000ms）

汇编语言程序设计实验报告

由于本次实验中除以 128 操作中, 128 是 2^7 , 所以可以直接将除法操作更改成右移七位, 这不仅仅把 `idiv` 这一消耗较高的指令优化掉, 同时也减少了中间寄存器的使用, 减少了两条 `mov` 语句的使用, 不过在这里这样的优化其实提升几乎是没用的, 一方面是除法指令使用次数不多, 另一方面是除数较小。

优化前运行结果如图 1.15 所示。

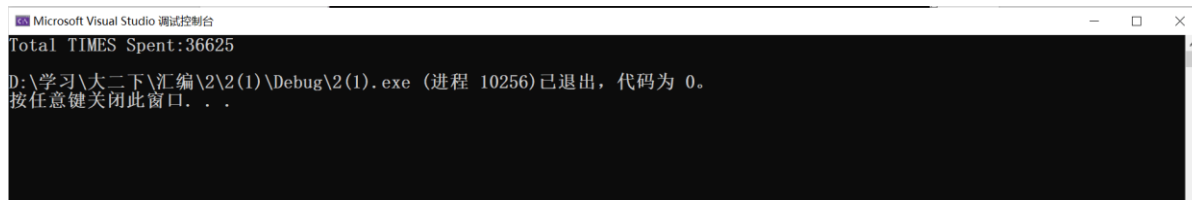


图 1.15 优化前运行结果

优化除法后运行结果如图 1.16 所示。

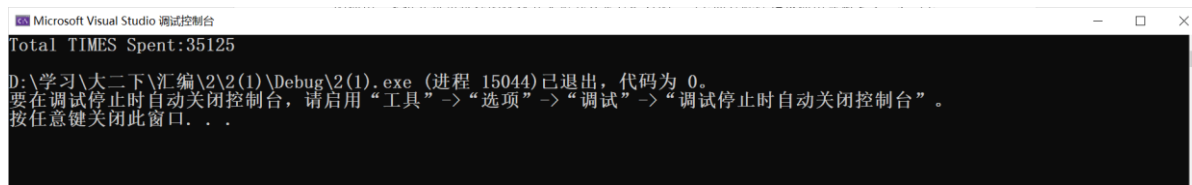


图 1.16 优化除法运行结果

此外, 乘法其实也可以用移位操作进行优化, 例如此处的乘 5, 由于 $5 = 2^2 + 2^0$, 因此 $a*5$ 可以改写成 $(a<<2)+a$, 但是显而易见, 这样改写乘法操作在这里很有可能没有任何优化, 这是因为这里的 5 本身是一个很小的数, 另外如果使用这种改写方法的话需要增加 `mov` 指令, 也就是增加了操作数。运行结果如图 1.17 所示。

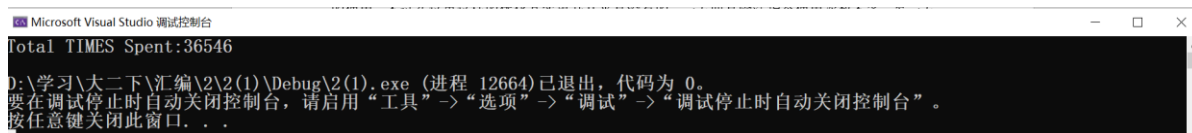


图 1.17 优化乘法后运行结果

显然这里发生了负优化, 这也是在我们的意料之中的。

优化二: 优化拷贝方式

正如上述的讨论, 这里的优化应当为整个程序带来极高的性能提升, 因为这大幅减少了整个过程中的指令数。此外, 这里直接将上文中的优化方法二、三一起进行, 因为个人认为这两种优化方法的思想实质上是一致的, 即通过循环展开来减少指令数, 进而优化整个程序的性能。

在优化前, 整个拷贝过程如图所示, 可以发现逐字节拷贝在循环过程中的指令数明显偏多。相关代码如图 1.18 所示。

```
L3:
    mov bl,(BYTE ptr Init[ecx])
    mov (BYTE ptr MIDF[esi]),bl
    inc esi
    inc ecx
    inc edx
    cmp edx,TYPE S
    jna L3
    jmp L5
```

图 1.18 逐字节拷贝部分代码

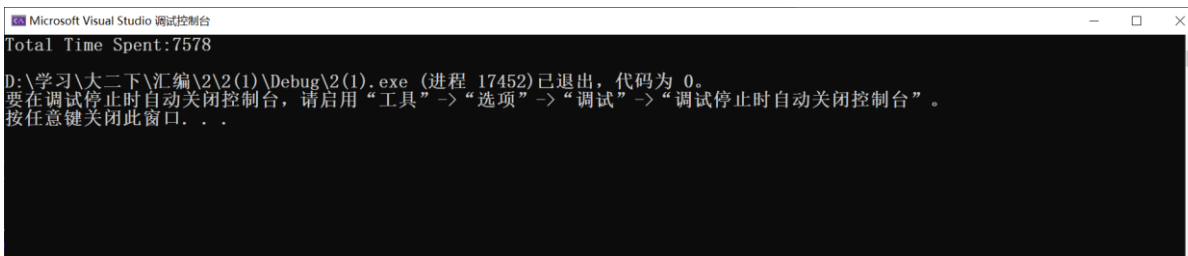
汇编语言程序设计实验报告

在优化后，直接将拷贝时的小循环优化掉，指令数大幅减少。相关代码如图 1.19 所示。

```
L2:
mov ebx,(DWORD ptr Init[ecx].SAMID[0])
mov (DWORD ptr LOWF[ebp].SAMID[0]),ebx
mov ebx,(DWORD ptr Init[ecx].SAMID[4])
mov (DWORD ptr LOWF[ebp].SAMID[4]),ebx
mov bl,(BYTE ptr Init[ecx].SAMID[8])
mov (BYTE ptr LOWF[ebp].SAMID[8]),bl
mov ebx,Init[ecx].SDA
mov LOWF[ebp].SDA,ebx
mov ebx,Init[ecx].SDB
mov LOWF[ebp].SDB,ebx
mov ebx,Init[ecx].SDC
mov LOWF[ebp].SDC,ebx
mov ebx,Init[ecx].SF
mov LOWF[ebp].SF,ebx
add ebp,TYPE S
jmp L5
```

图 1.19 优化拷贝方式后的部分代码

优化后运行时间如图 1.20 所示。



Microsoft Visual Studio 调试控制台
Total Time Spent: 7578
D:\学习\大二下\汇编\2\2(1)\Debug\2(1).exe (进程 17452) 已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。...

图 1.20 优化拷贝方式后的运行时间

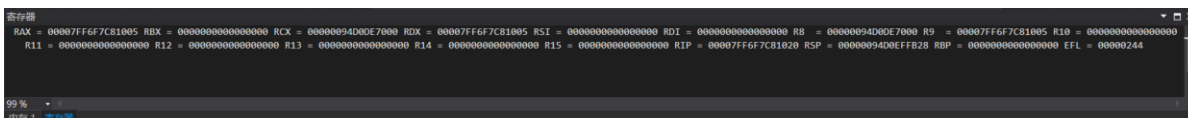
2. 不同约束条件、算法与程序结构带来的差异

通过上文的测试以及后续在实验五中的测试，我们可以看到通过从逐字节拷贝依次更换为逐字拷贝、逐多字拷贝，从算法的方面进行循环展开，尽可能减小关键路径的长度，就能使程序具有更好的性能，其关键就在于尽可能减少循环的次数。经过查阅资料发现，这样的原因在于 CPU 内部在运行程序的时候会提供一个并行化计算，而对整个程序进行循环展开之后，CPU 可以通过这种简单的并行去执行整个程序，进而决定整个程序运行时间的就是程序中的关键路径。

3. 几种编程环境中程序的特点记录与分析

本次实验首先使用 vs2019 编写调试了 32 位程序，后续使用 DosBox 和 QEMU 下 ARMv8 的工具包来调试程序并熟悉这两种环境的特点，并且使用 vs2019 调试了 64 位程序。

对于 32 位和 64 位程序的差别，首先从源代码就可以看出来寄存器发生了变化，从原来 32 位的寄存器如 eax 进行了相应的位扩展，变成了如 rax 的形式，此外 64 位程序也增加了形如 r8、r9 这样的寄存器，此外通过 vs2019 进行单步调试可以发现地址和寄存器全部是 64 位的。如图 1.21 所示。



寄存器
RAX = 00007FF6F7C81005 RBX = 0000000000000000 RCX = 00000094D8DE7000 RDX = 00007FF6F7C81005 RSI = 0000000000000000 RDI = 0000000000000000 R8 = 00000094D8DE7000 R9 = 00007FF6F7C81005 R10 = 0000000000000000
R11 = 0000000000000000 R12 = 0000000000000000 R13 = 0000000000000000 R14 = 0000000000000000 R15 = 0000000000000000 RIP = 00007FF6F7C81020 RSP = 00000094D8DEFFB28 RBP = 0000000000000000 EFL = 00000244
99% -
内存 1 寄存器

图 1.21 调试 64 位程序时的寄存器窗口

对于 QEMU 下 ARMv8 的工具包，ARM 指令与 80X86 指令在格式上已经有了比较大的区别，具体可以参考 3.2.3，这里不再赘述。

汇编语言程序设计实验报告

1.5 小结

1.5.1 主要认识与收获

首先是对于汇编语言的总体有了较为直观深刻的理解,使用寄存器而不是变量来存一些频繁访问的变量可以显著提升程序的速度,这个之前很早就知道了,但是只有认真编过汇编程序,认真写过相应的代码才能更好的明白为什么。

其次是一些知识点与细节的收获,一些细节比如说宏重复调用会报错要使用 `LOCAL` 来解决、`printf` 可能会改变作为参数的寄存器变量,变址寻址可以减少指令的数目等等这些敲完代码踩过坑甚至花了很长时间才解决的问题印象才最深刻。

然后是 Debug 的一些收获,之前对于 VS2019 的 memory 窗口一直不理解一堆 01 什么的都是干嘛的,这里第一个实验就学到了这些表示的内存的存储的值,课本上学过的小端存储、内存地址在这里看得非常直观。

1.5.2 思考题

实验三任务 3.1 思考题:

1. 子程序与主程序之间是如何传递信息的?刚进入堆栈时,堆栈栈顶及之下存放了一些什么信息?执行 `CALL` 指令及 `RET` 指令,CPU 完成了哪些操作?若执行 `RET` 前把栈顶的数值改掉,那么 `RET` 执行后程序返回到何处? `invoke` 伪指令对应的汇编语句有哪些?子程序中的局部变量的存储空间在什么位置?如何确定局部变量的地址?访问局部变量时的地址表达式有何特点?

答:在主程序调用子程序时,通过观察 `invoke` 指令的反汇编代码,发现分为两步,首先将参数压栈,接着调用 `call` 跳转到子程序处,并且把 `invoke` 指令的下一条指令的操作地址压栈。此时堆栈下存放了操作地址和参数。执行 `ret` 指令时,将操作地址出栈,并跳转到该指令。如果将栈顶数据改掉,就会跳转到变化后的操作地址。`Invoke` 对应的汇编语句有 `push` 和 `call`。在进入子程序的时候,通过修改堆栈指针 `esp` 来预留出需要的空间,在用 `ret` 指令返回主程序之前,同样通过恢复 `esp` 丢弃这些空间,这些变量就随之无效了。可以通过 `esp` 和局部变量的定义顺序确定地址。通常使用变址寻址,如`[esp+4]`来访问局部变量。

2. 观察模块间的参数的传递方法,包括公共符号的定义和外部符号的引用,若符号名不一致或类型不一致会有什么现象发生?

答:通过将参数压栈来传递,如果符号名不一致或类型不一致会报错。

实验三任务 3.2 思考题:

1. (地址)类型转换的含义是什么? (比如, `char a[10];` 一种地址类型转换的做法: `*(int *)a=123;`)

答:用另一种方式去看地址中数据表示的方式。

2. C 语言函数调用语句对应的汇编语句有哪些?调用函数与被调用函数之间是如何传递信息的?如果汇编语言子程序的语言风格不是 C 语言风格,被 C 语言程序调用时是否会出错?

答:如果调用函数有参数则对应 `push` 和 `call`,否则对应 `call`。通过将参数和返回地址压栈来传递信息。不会出错。

汇 编 语 言 程 序 设 计 实 验 报 告

1.5.3 实验操作的经验教训

1. 宏重复调用的时候要使用 LOCAL 来解决
2. 多文件调用的时候记得 public 和 extern，否则编译的时候会找不到 external obj 而报错
3. 调用函数时有可能会改变对应寄存器的值所以记得及时进行 pop push 栈操作来保护现场
4. 使用子程序的时候注意 esp 的位置和恢复其指向，另外子程序记得及时 push 保护一些寄存器的值。

二、利用汇编语言特点的实验

2.1 目的与要求

掌握编写、调试汇编语言程序的基本方法与技术，能根据实验任务要求,设计出较充分利用了汇编语言优势的软件功能部件或软件系统。

2.2 实验内容

在编写的程序中，通过加入内存操控，反跟踪，中断处理，指令优化，程序结构调整等实践内容，达到特殊的效果。

2.3 实验过程

2.3.1 实验方法说明

1. 中断处理程序的设计思想与实验方法

(1) 实验要求

在 DOSBox 环境中实现时分秒信息在窗口指定位置的显示。其中，指定位置信息来源于程序中定义的变量的内容；所实现的程序运行后需要驻留退出，并能避免重复安装。

要求能在 TD 下观察中断矢量表、观察已有的某个中断处理程序的代码、读取 CMOS 中某个单元内容；能通过某种方式在 TD 下调试中断处理程序。

(2) 实验基本思路

主程序处理流程：1. 检查避免重复安装：每次运行时判断当前 08H 的存放内容，如果是原来的 08H 的地址那么程序正常运行，否则就不再继续运行程序。2. 安装中断处理程序 3. 驻留退出程序：等待用户按下“q”，程序会恢复中断矢量然后正常退出。中断矢量表可以通过 TD 调试工具来进行观察中断矢量表。

08 中断处理子程序流程：注意虽然可以使用通过 I/O 端口(70H/71H)来直接读取“实时时钟/系统配置接口芯片(RT/CMOS RAM)”的内容从而来获取我们的时间，这比使用“INT 1AH”这种方法要快一些，可以提高程序运行效率。而在 RT/CMOS RAM 的存储单元中，当前时间信息存放的位置是：“时”的偏移值为 4，“分”的偏移值为 2，“秒”的偏移值为 0。同时需要说明这些时间信息是按照 BCD 码的形式进行存储。这里每次中断运行时首先会调用原来 8 号中断程序，而后对中断次数进行计数，倒计时从 18 见到 0 时返回同时给计数器赋初值。期间使用 I/O 操作获取当前的时间后转换为对应的字符串（人能读懂的信息）使用“INT 10H”来显示信息。然后中断返回程序结束。

2. 反跟踪程序的设计思想与实验方法

本次实验中主要尝试了如下几种反跟踪的方法：

(1) 根据实验指导书，尝试采取的第一种反追踪方案是在数据段内利用简单函数对密码进行映射处理，防止反汇编时直接通过查看数据段得到密码。这里采用的加密函数为 $(X-20H)*3$ ，相关代码如图 2.1 所示。

汇编语言程序设计实验报告

```
password db ('1'-20h)*3
          db ('2'-20h)*3
          db ('3'-20h)*3
          db ('4'-20h)*3
          db ('5'-20h)*3
          db ('6'-20h)*3
          db 0
```

图 2.1 使用函数映射处理密码

(2) 根据实验指导书, 尝试建立地址表用于间接转移, 相关代码如图 2.2 所示。经过实践后发现 windows 下 VS2019 不允许使用类似于 jmp P1 的汇编代码进行分支跳转, 然后放弃使用。

```
mov P1,offset Pass
mov P2,offset L2
mov E1,offset exit
```

图 2.2 最初建立的部分地址表

(3) 根据实验指导书, 尝试通过计时来防止通过单步调试来破解程序的可能性。在这里计时通过调用 GetTickCount 函数实现, 相关代码如图 2.3 所示。

```
invoke GetTickCount
mov startTime,eax
mov eax,0123H
mov ebx,0456H
add ax,bx
add ax,ax
mov ax,4c00H
invoke GetTickCount
```

图 2.3 通过计时防止单步调试的部分代码

(4) 根据实验指导书, 也可以考虑增加无关代码, 以增大破解难度。这一部分具体就是在整个程序的各个地方增加一些无用的指令, 来增大破解难度。

(5) 根据实验指导书, 也可以通过检查堆栈来抵制动态调试跟踪, 即判断单步调试时是否修改了堆栈段的内容, 相关代码如图 2.4 所示。

```
OK:
  push offset P0
  pop ecx
  mov ecx,ebx
  add ecx,ecx
  mov ebx,[esp-4]
  jmp ebx
```

图 2.4 通过检查堆栈防止单步调试的代码

3. 指令优化及程序结构的实验方法

实验 2 中主要使用了两种优化方法: 乘除操作改为移位操作以及使用循环展开的思路来优化指令个数。具体分析请参见 1.4 小节。

实验 3 主要就是宏的使用以及分模块、子程序的使用。在实现过程中应注意模块之间的连接, 合适的模块化编程可以大幅降低系统的耦合性。

2.3.2 实验记录与分析

1. 中断处理程序的特别之处

汇编语言程序设计实验报告

(1) 更改时钟位置

首先通过修改 DH/DL 来更改时钟的位置，相关代码如图 2.5 所示。

```
; MOV DH,0 ; 显示在 0 行  
; MOV DL,80 -BUF_LEN ; 显示在最后几列(光标位置设到右上角)  
MOV DH,YLOC  
MOV DL,XLOC
```

图 2.5 更改时钟位置的代码

通过给出的源程序以及注释可以看出这里规定了时钟显示的位置，因此这里采取同样的方法将时钟位置设置在左上角，这里的 YLOC 与 XLOC 均为在数据段定义的变量，前者为 1，后者为 2。此外，在这里也通过更改 BL 来改变了程序的底色，将其变成白色，

在 DOSBOX 中运行程序，可以看到在左上角已经出现了当前的时间，运行结果如图 2.6 所示。

```
Z:\>mount c: d:\x86\masm60  
Dr17:48:58 mounted as local directory d:\x86\masm60\  
  
Z:\>c:  
  
C:\>masm TEST2.ASM  
Microsoft (R) MASM Compatibility Driver  
Copyright (C) Microsoft Corp 1991. All rights reserved.  
  
Invoking: ML.EXE /I. /Zm /c /Ta TEST2.ASM  
  
Microsoft (R) Macro Assembler Version 6.00  
Copyright (C) Microsoft Corp 1981-1991. All rights reserved.  
  
Assembling: TEST2.ASM  
  
C:\>LINK TEST2.OBJ;  
  
Microsoft (R) Segmented Executable Linker Version 5.20.034 May 24 1991  
Copyright (C) Microsoft Corp 1984-1991. All rights reserved.  
  
C:\>TEST2
```

图 2.6 在 DOSBOX 中的运行结果。

(2) TD 观察中断矢量表

首先通过 TD.exe 进入中断矢量表之后通过 goto 转到相应的 int21 号矢量表的位置，相关操作如图 2.7 所示。

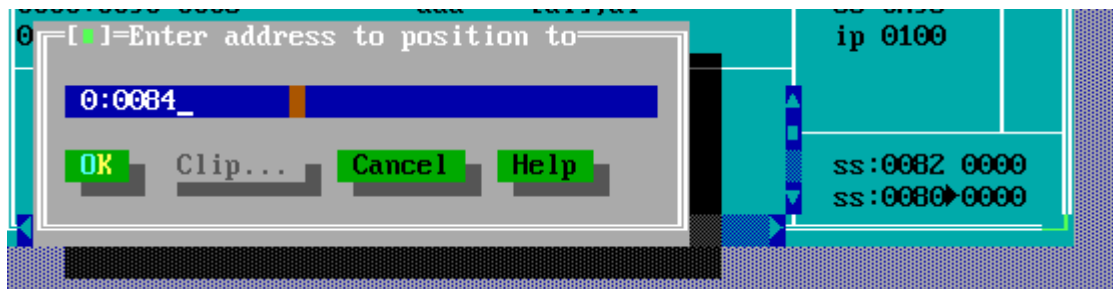


图 2.7 通过 goto 语句跳转的过程

然后就可以得到对应的 8 号中断程序对应的地址 0553:00CA，其结果如图 2.8 所示。


```
fs:0084 CA 00 53 05 10 08 98 01  Ssp  j
fs:008C 22 08 53 05 70 08 53 05  Ssp S
fs:0094 C0 04 00 F1 E0 04 00 F1  L ±α± ±
fs:009C 00 05 00 F1 20 05 00 F1  ± ± ± ±
```

接下来就可以根据对应的位置进行跳转，跳转后的结果如图 2.9 所示。

0553:00CA	9C	pushf
0553:00CB	80FC4C	cmp ah,4C
0553:00CE	7521	jne 00F1
0553:00D0	1E	push ds
0553:00D1	50	push ax
0553:00D2	53	push bx

在 TD 中通过单步调试，并且再遇到 INT 21H 中断程序时，通过 Alt+F7 进入中断程序。此时 cs 为 0553，ip 为 00CA，如图 2.10 所示，此时再通过从中断矢量表中找到 INT 21H 中断，中断程序地址与前面一样，均为 0553:00CA，如图 2.11 所示。

```

[ ]-CPU 80486
cs:00CA 9C      pushf
cs:00CB 80FC4C   cmp     ah,4C
cs:00CE 7521     jne     00F1
cs:00D0 1E      push    ds
cs:00D1 50      push    ax
cs:00D2 53      push    bx
cs:00D3 B83602   mov     ax,0236
cs:00D6 8ED8     mov     ds,ax
cs:00D8 B451     mov     ah,51
cs:00DA CD21     int     21
cs:00DC 2E3B1E6E03 cmp     bx,cs:[036E]
cs:00E1 7405     je      00E8
cs:00E3 5B      pop     bx

es:0000 CD 20 E4 9D 00 EA FF FF = Σ¥ Ω
es:0008 AD DE 32 0B FA 07 22 08 i 2δ.·"
es:0010 53 05 70 08 53 05 7B 01 SαP$+Ⓢ
es:0018 01 01 01 00 02 FF FF FF 000 0

ax 3508    c=0
bx 0000    z=0
cx 0000    s=0
dx 0000    o=0
si 0000    p=0
di 0000    a=0
bp 0000    i=0
sp 00C2    d=0
ds 0ABD
es 0AA0
ss 0AB0
cs 0553
ip 00CA

ss:00C4 0ABD
ss:00C2 00A9

```

```

[ ]=CPU 80486                                     1= [ ] [ ]
cs:00CA 9C      pushf                                ax 3508      c=0
cs:00CB 80FC4C   cmp      ah,4C                     bx 0000      z=0
cs:00CE 7521     jne      00F1                       cx 0000      s=0
cs:00D0 1E       push    ds                          dx 0000      o=0
cs:00D1 50       push    ax                          si 0000      p=0
cs:00D2 53       push    bx                          di 0000      a=0
cs:00D3 B83602   mov     ax,0236                     bp 0000      i=0
cs:00D6 8ED8     mov     ds,ax                        sp 00C2      d=0
cs:00D8 B451     mov     ah,51                       ds 0ABD
cs:00DA CD21     int     21                           es 0AA0
cs:00DC 2E3B1E6E03 cmp    bx,cs:[036E]              ss 0AB0
cs:00E1 7405     je      00E8                         cs 0553
cs:00E3 5B       pop     bx                          ip 00CA

fs:0084 CA 00 53 05 10 08 98 01  11 S< 1j0
fs:008C 22 08 53 05 70 08 53 05  " S< p S<
fs:0094 C0 04 00 F1 E0 04 00 F1  L< ± ± ±
fs:009C 00 05 00 F1 20 05 00 F1  ± ± ± ±

```

2. 反跟踪效果的验证

汇编语言程序设计实验报告

在这一环节中，考虑使用 vs2019 的单步调试功能来进行测试，此外这里重点讨论使用计时法和检查堆栈法来防止单步调试的有效性，对于其余的几种方法这里就不再赘述。

(1) 计时法:

首先考虑在最开始调用 GetTickCount 时打一个断点，然后开始单步调试，单步调试开始时的截图如图 2.12 所示。

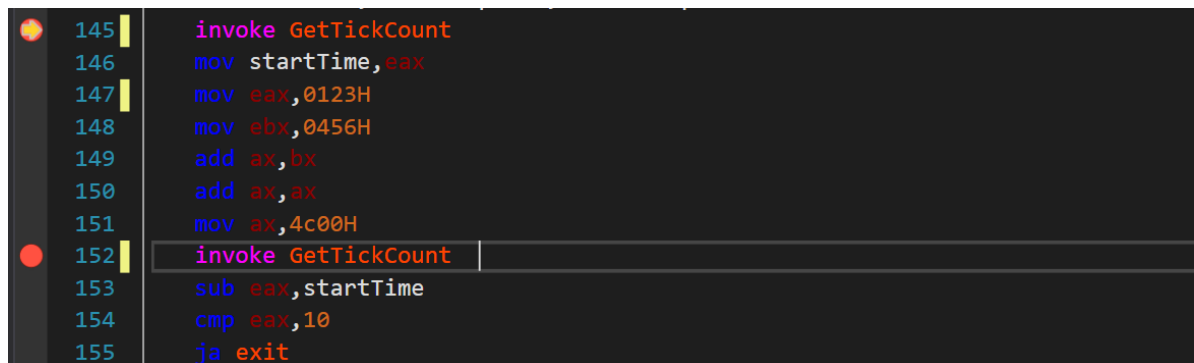


图 2.12 单步调试开始时截图

然后在第 154 行打断点，看一下此时 eax 寄存器的值，因为此时 eax 中的值就是运行的时间，监视窗口如图 2.13 所示。

名称	值	类型
eax	7625	unsigned int

图 2.13 eax 寄存器的值

发现 eax 的值为 7625，随后由于时间过长就会跳转到程序终止的位置，这就说明计时法有效，可以成功防止单步调试。

(2) 检查堆栈法:

这里主要是在计时法通过之后进行的测试，可以通过单步调试检查是否有效。我们在将地址入栈的时候加入断点，然后单步调试，可以看到，标号 P0 的偏移地址 0x00ef85eb 已经入栈，此时内存窗口如图 2.14 所示。

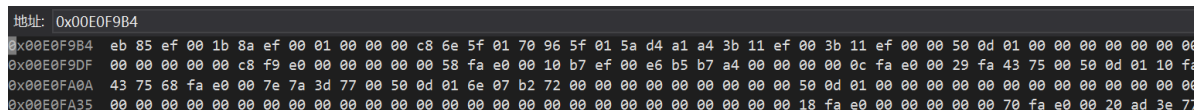


图 2.14 将 P0 偏移地址入栈后的内存窗口

然后继续单步调试，出栈后再取对应位置的地址，在程序中将该内容存到了寄存器 ebx 中，通过查看 ebx 的值，发现依旧为 0x00ef85eb，运行结果如图 2.15 所示，则检查堆栈法失效，失效原因应该是 VS2019 工具在单步调试时并不修改当前程序的堆栈段。

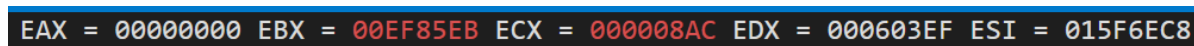


图 2.15 此时寄存器窗口的值

3. 跟踪与破解程序

我和张贺圆、吴鼎三人合作，由我破解张贺圆的程序。

首先，由于之前使用过 objdump 进行过逆向工程分析，所以这次最开始就是考虑使用 objdump 进行反汇编，但是实际运行之后发现效果并不好，原有的符号名它也没能保存，并且它似乎并没有反汇编成 intel 格式的代码，而是 ATT 格式的汇编指令，遂放弃使用，部分代码如图 2.16 所示。

汇编语言程序设计实验报告

```
409120: cc int3
409121: cc int3
409122: 33 c0 xor %eax,%eax
409124: 39 05 20 b3 47 00 cmp %eax,0x47b320
40912a: 0f 94 c0 sete %al
40912d: c3 ret
40912e: cc int3
40912f: cc int3
409130: cc int3
409131: b8 60 d8 47 00 mov $0x47d860,%eax
409136: c3 ret
409137: b8 54 d8 47 00 mov $0x47d854,%eax
40913c: c3 ret
40913d: 55 push %ebp
40913e: 8b ec mov %esp,%ebp
409140: 81 ec 24 03 00 00 sub $0x324,%esp
409146: 53 push %ebx
409147: 6a 17 push $0x17
409149: ff 15 30 e0 47 00 call *0x47e030
40914f: 85 c0 test %eax,%eax
```

图 2.16 利用 objdump 反汇编得到的代码

后来听说了 cutter 这个工具，就去尝试了一下，个人感觉界面风格简洁，并且功能也比较强大。

首先将该可执行目标文件放入 cutter 中，很快 cutter 给出了该文件的种种信息，我们在反汇编代码框中全局搜索 main 函数，可以看出这里应当就是程序的入口，如图 2.17 所示。

```
int main(int argc, char **argv, char **envp):
0x00408507 mov dword [0x47b28b], 0x408608
0x00408511 mov dword [0x47b28f], 0x408670
0x0040851b mov dword [0x47b293], 0x408749
0x00408525 push str.please_enter_your_username ; 0x47b229 ; int32_t arg_ch
0x0040852a push str.s ; section..data
; 0x47b000 ; int32_t arg_8h
0x0040852f call fcn.00401bea
0x00408534 add esp, 8
0x00408537 push 0x47b281 ; int32_t arg_ch
0x0040853c push 0x47b005 ; int32_t arg_8h
0x00408541 call fcn.004011c2
0x00408546 add esp, 8
0x00408549 push eax
0x0040854a push ebx
0x0040854b mov bh, 0
0x0040854d mov eax, 0
0x00408552 mov bh, byte [eax + 0x47b266]
0x00408558 cmp bh, byte [eax + 0x47b281]
0x0040855e jne 0x40856f
0x00408560 inc eax
0x00408561 cmp bh, 0
0x00408564 jne 0x408552
0x00408566 mov byte [0x47b27f], 1
0x0040856d jmp 0x408576
0x0040856f mov byte [0x47b27f], 0
0x00408576 pop ebx
0x00408577 pop eax
0x00408578 cmp byte [0x47b27f], 1
0x0040857f jne 0x408646
0x00408585 push str.please_enter_your_password ; 0x47b244 ; int32_t arg_ch
0x0040858a push str.s ; section..data
; 0x47b000 ; int32_t arg_8h
0x0040858f call fcn.00401bea
0x00408594 add esp, 8
0x00408597 push 0x47b281 ; int32_t arg_ch
0x0040859c push 0x47b005 ; int32_t arg_8h
0x004085a1 call fcn.004011c2
0x004085a6 add esp, 8
0x004085a9 call sub.KERNEL32.dll_GetTickCount ; DWORD GetTickCount(void)
0x004085ae mov dword [0x47b031], eax
0x004085b3 mov eax, 0x123 ; 291
0x004085b8 mov ebx, 0x456 ; 1110
0x004085bd add ax, bx
0x004085c0 add ax, ax
0x004085c3 mov ax, 0x4c00
```

图 2.17 反汇编得到的 main 函数附近代码

图中蕴含了很多信息，首先最左边的黄色的线描述了跳转的逻辑，然后在这张图片的代码中，我们可以看到内存 0x408525 和内存 0x408585 分别将两个字符串压栈，并且我们可以看出这个字符串的内容就是提示语句，进而可以推断出内存 0x40852f 调用了 printf 函数用来提示用户输入，那么内存 0x408541 应当就是调用的 scanf 函数，此外我也注意到在内存 0x4085a9 处调用了函数

汇编语言程序设计实验报告

GetTickCount, 那么基本可以断定该同学采用了计时的方法来防止单步调试。

我们现在先来看输入用户名后的这一段逻辑, 尝试破解这位同学的程序的用户名, 代码如下图 2.18 所示。

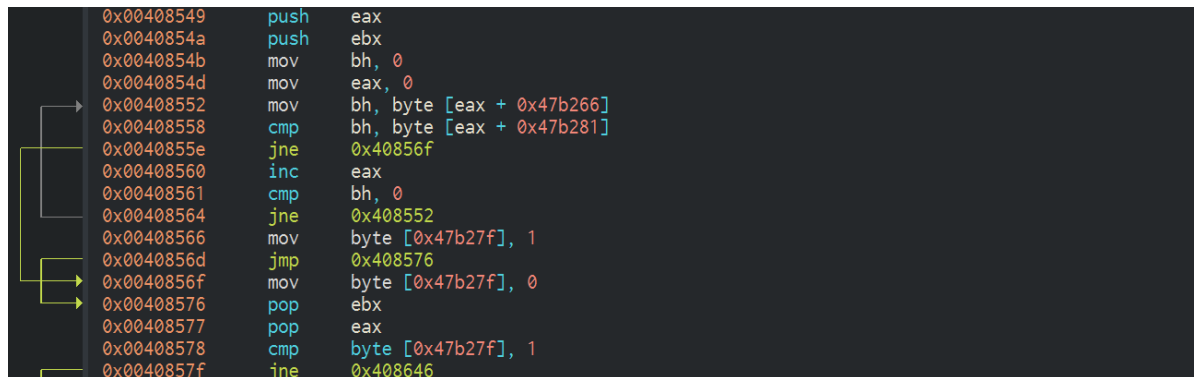


图 2.18 输入用户名后的部分反汇编代码

首先忽略掉保护现场的逻辑, 目测这里的 `eax` 应当是作为计数器, 而 `bh` 应当是作为一个比较的中间寄存器使用的, 此外, 这里涉及到了两个地址, 即 `0x47b266` 和 `0x47b281`, 这两个地址应该一个是储存正确的用户名, 而一个是用户名的地址, 结合上面调用 `scanf` 函数时候压栈的参数, 可以发现 `0x47b281` 应当是我们输入的字符串, 所以 `0x47b266` 就是正确的用户名。然后我发现这个软件可以进行简单的调试, 但是我没有找到如何用这个软件查看内存地址的值, 但是这个程序中借助了 `bh` 寄存器来作为中间变量, 而且这里还是把正确的答案先赋给 `bh` 然后再和输入的字符串进行比较, 如果这里是反着来的, 我可能还需要花费一些精力想想如何读取内存地址。我们在 `0x408552` 加断点, 看一下寄存器 `bh` 的值, 就可以知道用户名, 下面给出第一次比较时候的寄存器值, 如图 2.19 所示。

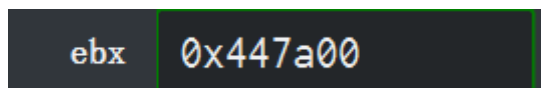


图 2.19 ebx 的值

对照 `ascii` 码表, 可以知道 `0x7a` 对应小写字母 `z`, 那这个时候基本就可以确定用户名就是 `zhy` 了。

接下来我们尝试去破解密码。首先我们看一下在输入用户名之后, 他应该进行了相应的判断, 也就是说如果用户名不对的话它按照要求会去重复以上过程三次, 否则退出程序。接下来的一部分代码应该就是关于比对密码是否正确的部分了, 如图 2.20 所示

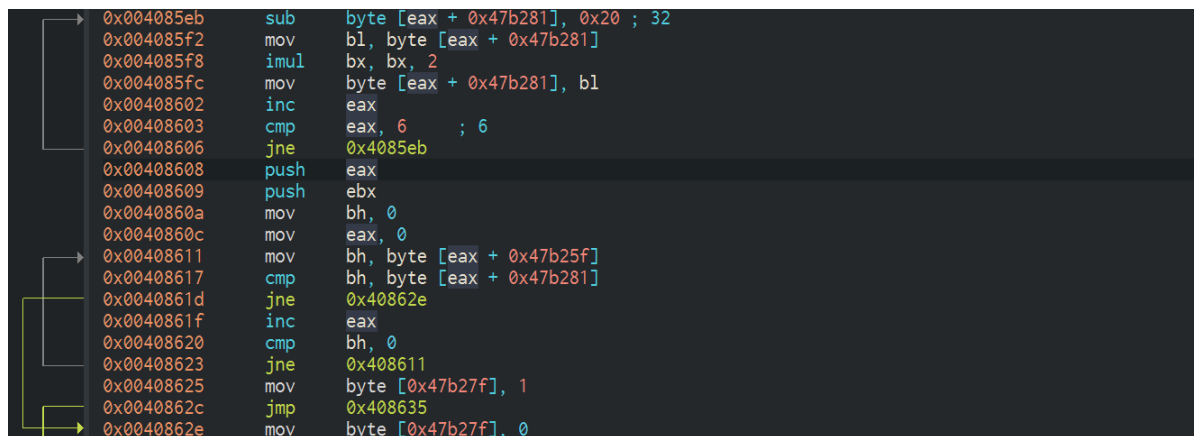


图 2.20 比对密码反汇编代码

汇编语言程序设计实验报告

发现我又看到了 0x47b281 这一个地址，结合输入密码时采用堆栈传参时压栈的地址，可以断定 0x47b281 就是我们输入的密码的地址。然后我们可以看到，他这里对这个地址的值进行了一系列操作，应该就是用了一个函数对密码进行映射，所以这里需要对输入进行相应的映射，通过这一串代码可以看出来这个映射函数应当是 $(x-32)*2$ ，然后后面的 `eax` 应该是作为一个计数器使用的，所以可以认为密码的长度应该是 6，根据内存 0x408611 处的比较，可以断定密码就在 0x47b25f 处，那其实我们只需要看一下这个内存地址的值即可。而且到这里，我惊讶的发现他竟然在这里没有设置计时器来组织单步调试，这也就意味着我直接在这里打断点然后看一下 `bh` 的值就可以知道密码的第一位了，依次类推可以得到整个密码。我们仅以第一位为例来破译，下图 2.21 给出了执行到 0x408611 处的 `ebx` 的值，那么对应的 `bh` 的值就是 0x22，进而可以知道第一位的 `ascii` 码为 1。

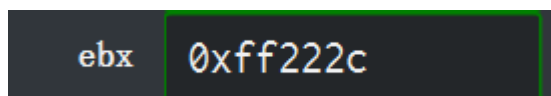


图 2.21 `ebx` 的值

最终，我成功破译了这个程序，如图 2.22 所示。

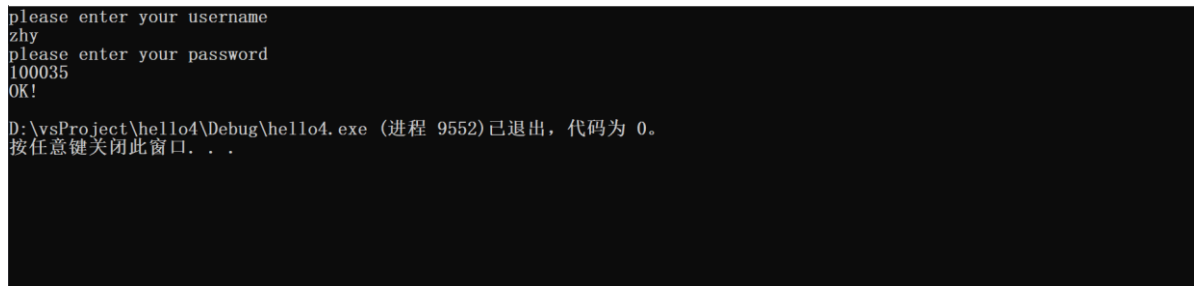


图 2.22 最终破译成功的示意图

4. 特定指令及程序结构的效果

实验 2 中主要就是两种优化方法：乘除操作改为移位操作以及利用循环展开优化拷贝方式来减少指令数。具体分析请参见 1.4 小节。

实验 3 中 `invoke` 外部模块自己定义的函数传参可以不借助形参，只需要借助约定的寄存器传参即可。

2.4 小结

2.4.1 认识与收获

(1) 系统中断处理程序

在此实验中我学会了通过更改有关寄存器来更改显示时钟在 `DosBox` 中的位置，同时也经过不断的尝试与向老师、同学请教，搞懂了中断矢量表本身的含义，也在 `DosBox` 的 `TD` 调试中一步一步找到了对应的访问方法。

(2) 加密与解密实验

通过本次实验，我对于汇编语言相对于高级语言的特点有了更深一步的了解，同时掌握了多种反追踪的方法，以及通过设置断点修改 `EIP` 等对程序进行解密的办法。因为汇编语言更接近机器指令，相较于 `C` 语言，汇编更能在机器层面上对程序进行操作。比如在加密的过程中，可以通过间接地址、动态修改代码、计时等方法来反追踪和反单步调试，在解密的过程中，可以先使用 `VS2019` 软件对可执行文件进行反汇编处理，通过阅读反汇编代码，尝试通过设置断

汇编语言程序设计实验报告

点和改变 EIP 的值跳过某些反追踪和反单步调试的程序。此外，这也启发了我，可以通过在 C 语言中内嵌汇编代码来对底层进行操作，从而更好的保护程序数据。

2.4.2 思考题

1. 思考一下，如何用 C 语言（不嵌入汇编语言）实现反跟踪？是否能发现汇编语言的特殊之处？

答：首先，可以通过计时防止单步调试，这一点和在汇编语言中的实现方法类似，在关键代码部分调用 `GetTickCount` 来记录这段代码的执行时间，如果超过某个时间则说明有人试图在这段代码处进行单步调试，可以直接退出程序。此外，可以通过添加无关代码来扰乱实现。也可以通过利用一个函数对密码进行数学上的保护，来使其不那么容易被破解。但是用 C 语言实现的反跟踪的方法没有汇编语言那样丰富，最重要的原因就是汇编语言接近底层，可以直接对内存进行操作。

2.4.3 经验教训

首先我认识到在学习知识时总是需要借助搜索引擎去更好的学习的，不要仅仅局限于课本上的内容，如果仅仅局限于课本或是老师提供的资料，有时候不能很快的找到解决方法，还是要多依赖于自己手头的各类工具。另外就是调试的时候务必要彻底理解概念，因为如果看别人这样走了一遍后自己重新来还是很容易卡住，这是因为自己毕竟不太懂原理，我当时调 DOSBOX 的时候就是看别人调好像已经会了，但是到自己的时候很容易因为没有掌握原理然后就不知道为什么这样做或者下一步应该怎么做，直到后来完全理解后才明白可以顺畅的做出来。

在做实验的过程中也遇到了一些问题。首先就是在 windows 下的 32 位段系统中，有一些方法并不适用，比如设置地址表进行间接转移，而且在 VS2019 中，每次编译运行时的机器指令代码都不相同，导致无法通过在程序中采用直接转移的方式。另外对寻址方式的不熟悉也导致在编程过程中出现了不少语法错误，但都得以顺利解决。

在解密时，面对很多反汇编代码有点茫然无措，但是静下心来好好分析之后就可以抽丝剥茧，一点点去尝试解密同学的程序。

三、工具环境的体验

3.1 目的与要求

熟悉支持汇编语言开发、调试以及软件反汇编的主流工具的功能、特点与局限性及使用方法。

3.2 实验过程

3.2.1 WINDOWS10 下 VS2019 等工具包

任务 1.1

(1) 显示反汇编窗口，了解 C 语言与汇编语句的对应关系。在反汇编窗口中的“查看选项”下有“显示符号名”，指出勾选与不勾选该项选时，反汇编窗口显示内容的差异；

答：是否勾选“显示符号名”选项的反汇编窗口分别如图 3.1 和图 3.2 所示。

```
00C818EC 8D 7D DC      lea     edi,[ebp-24h]
00C818EF B9 09 00 00 00  mov     ecx,9
00C818F4 B8 CC CC CC CC  mov     eax,0CCCCCCCCh
00C818F9 F3 AB          rep stos dword ptr es:[edi]
00C818FB A1 24 A0 C8 00  mov     eax,dword ptr [__security_cookie
00C81900 33 C5          xor     eax,ebp
00C81902 89 45 FC          mov     dword ptr [ebp-4],eax
00C81905 B9 03 C0 C8 00  mov     ecx,offset _0AE8B74C_源@cpp (0C8
00C8190A E8 0C FA FF FF  call    @__CheckForDebuggerJustMyCode@4
short z;
char str[10] = "The end!";
00C8190F A1 30 7B C8 00  mov     eax,dword ptr [string "The end!"
00C81914 89 45 E0          mov     dword ptr [str],eax
```

图 3.1 勾选“显示符号名”选项的反汇编窗口

```
00C818EC 8D 7D DC      lea     edi,[ebp-24h]
00C818EF B9 09 00 00 00  mov     ecx,9
00C818F4 B8 CC CC CC CC  mov     eax,0CCCCCCCCh
00C818F9 F3 AB          rep stos dword ptr es:[edi]
00C818FB A1 24 A0 C8 00  mov     eax,dword ptr ds:[00C8A024h]
00C81900 33 C5          xor     eax,ebp
00C81902 89 45 FC          mov     dword ptr [ebp-4],eax
00C81905 B9 03 C0 C8 00  mov     ecx,0C8C003h
00C8190A E8 0C FA FF FF  call    00C8131B
short z;
char str[10] = "The end!";
00C8190F A1 30 7B C8 00  mov     eax,dword ptr ds:[00C87B30h]
00C81914 89 45 E0          mov     dword ptr [ebp-20h],eax
00C81917 8B 0D 34 7B C8 00  mov     ecx,dword ptr ds:[00C87B34h]
```

图 3.2 不勾选“显示符号名”的反汇编窗口

若勾选“显示符号名”，则在“反汇编”窗口中会以符号的形式显示全局变量、局部变量。若不勾选此项，则显示的是该变量对应的地址。并且可以观察到，全局变量和局部变量显示的结果是

汇编语言程序设计实验报告

不同的。该显示结果也表明，变量名是一个地址的符号表示。全局变量对应的是一个段及段内偏移，局部变量对应的是堆栈中的一个存储单元。

(2) 显示寄存器窗口。在该窗口中设置显示 寄存器、段寄存器、标志寄存器等：

答：寄存器窗口如图 3.3 所示。

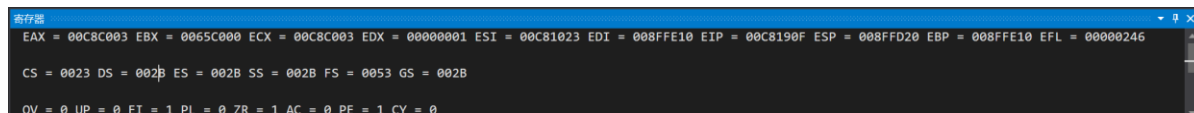


图 3.3 寄存器窗口

(3) 显示监视窗口，观察变量的值；显示内存窗口，观察变量的值（整型值、字符串等）在内存中的具体表现细节。

答：监视窗口如图 3.4 所示。

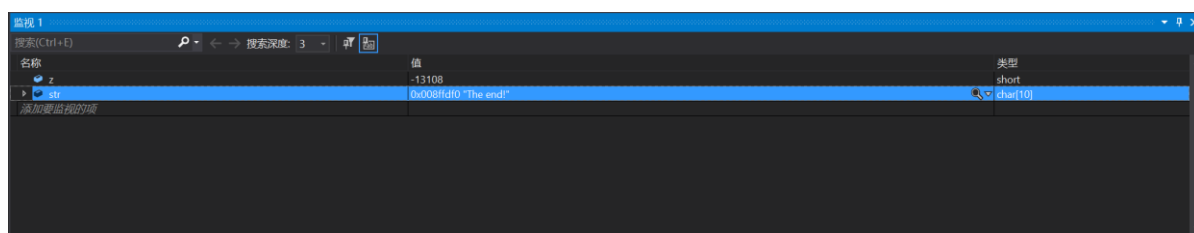


图 3.4 监视窗口

通过此窗口可以看到程序中未初始化的整型变量 z 的值，以及已经初始化的字符串 str。

字符串 str 的内存窗口如图 3.5 所示。

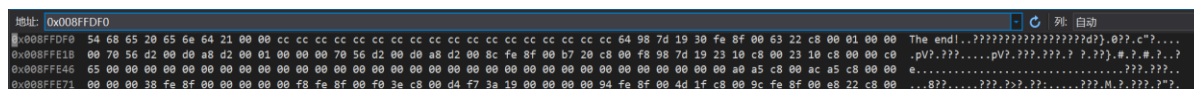


图 3.5 字符串 str 的内存窗口

可以观察到，54 是字符 'T' 的 ASCLL 码的 16 进制，68 是字符 'h' 的 ASCLL 码的 16 进制，以此类推，00 代表字符串的结束。

整型变量 z 的内存窗口如图 3.6 所示。

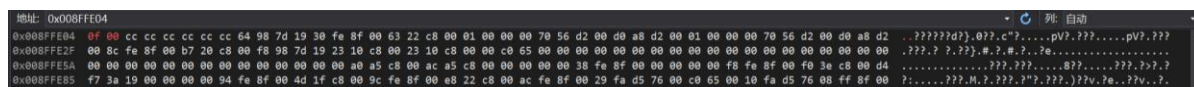


图 3.6 整型变量 z 的内存窗口

可以看出，此时 z 中的值正是 15。

同时也可以看一下全局变量中的数组 a 的存储方式。如图 3.7 所示。

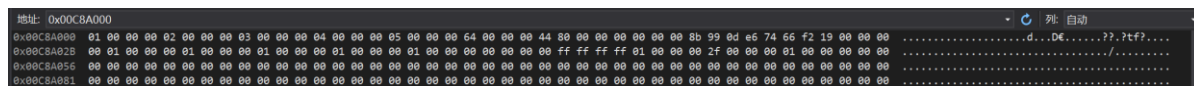


图 3.7 数组 a 的内存窗口

int 类型占 4 个字节，数组 a 有五个数，并且低位在前，高位在后。

(4) 有符号与无符号整型数是如何存储的；

答：首先观察一下全局变量中的有符号数 x，y，其实这里也可以参考图 27，因为在全局变量定义时先定义了数组 a，随后分别定义了 x 和 y，所以三者的地址是紧挨着的，从图中可以观察到 x=100 为正数，表示为 64H，而 y=-32700 是负数，80 44H 是其补码表示。

另外可以观察无符号数 length 的内存窗口，如图 3.8 所示。

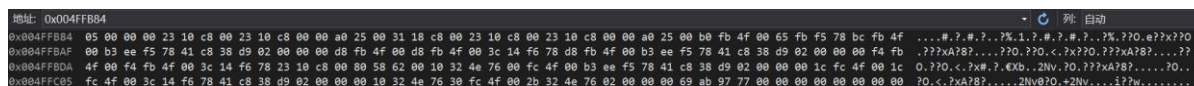


图 3.8 length 的内存窗口

汇编语言程序设计实验报告

可以看出存储方式并没有变化。

(5) 有符号数和无符号数的加减运算有无差别，是如何执行的？执行加法运算指令时，标志寄存器是如何设置的？执行比较指令时又有什么差异？

答：没有差别，执行过程中根据执行时的状态，设置相应的标志寄存器。在执行加法运算指令和比较运算指令时，EFL 的值在不断变化，但是这是由于运行过程中状态不断改变导致的。

(6) 观察思考：程序在编译时，在 sum 函数的 for (i = 0; i < length; i++) 处会给出警告信息：有符号/无符号不匹配。请问，i < length 最终采用的是有符号数比较还是无符号数比较？若将语句“z = sum(a, 5);”换成“z = sum(a, 0x90000000);”运行结果如何？为什么？如果将 sum 函数的参数 unsigned length 改为 int length，执行“z = sum(a, 0x90000000);”运行结果又如何？为什么？

答：最终采取的是无符号比较，这段代码的反汇编代码如图 3.9 所示。



图 3.9 比较时的反汇编代码

可以看出这里使用了 jae 命令，所以可以认为这里采用了无符号比较。

将“z=sum(a,5);”换成“z=sum(z,0x90000000);”会报错，因为这个时候数组会越界。

在此基础上，如果将 sum 函数的参数 unsigned length 改为 int length，则不会报错，但是这个时候也不会进入循环，因为此时 0x90000000 是一个负数。

任务 1.2

程序反汇编窗口如图 3.10 所示。

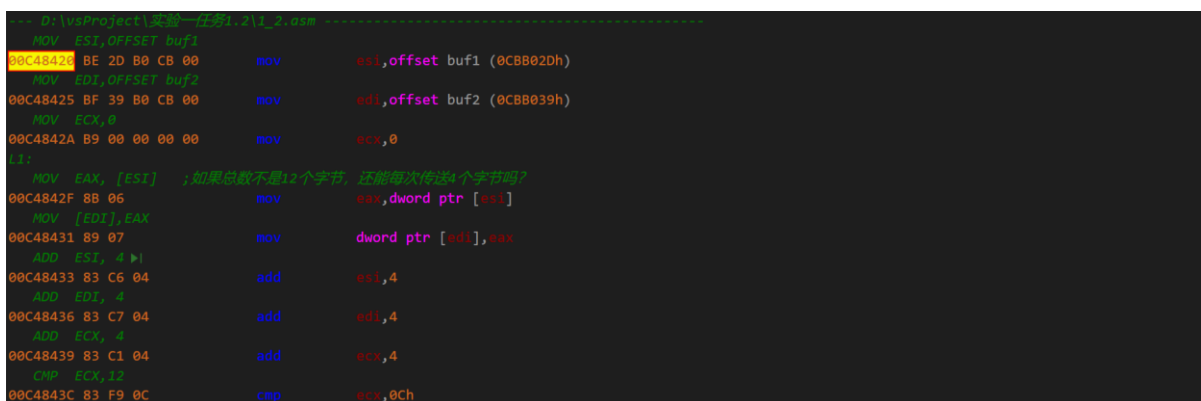


图 3.10 程序反汇编窗口

可以观察到，由目标代码反汇编得到的代码与原本的汇编程序的代码是相互对应的，有的汇编代码可以被翻译成一条指令，也可以被翻译成多条指令。

接下来可以看一下数据段中的内容，首先通过监视窗口可以查看 lpFmt 的地址为 0xcbb000，然后在内存窗口中查看相关信息，如图 3.11 所示。

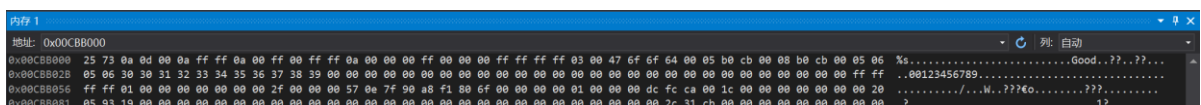


图 3.11 lpFmt 的内存窗口

可以看出lpFmt为25 73 0a 0d 00，后面是X，为：0a、ff、ff；其后紧跟着Y，为0a 00、ff 00、

汇编语言程序设计实验报告

ff ff; 其后紧跟着Z, 为0a 00 00 00、ff 00 00 00、ff ff ff ff; 其后紧跟着U, 03 00, 表明Z中有三个数据; 其后紧跟着STR1, 为47 6f 6f 64 00, 为Good的ASCLL码表示, 并且跟着0; 其后紧跟着P, 为05 b0 2e 00、08 b0 2e 00分别为X和Y的地址; 其后紧跟着Q, 为05 06 05 06, 即5 6重复两次; 其后紧跟着buf1, 为30 30 31 32 33 34 35 36 37 38 39 00, 即字符串"00123456789"; 其后紧跟着buf2, 为12个0。

接下来可以通过寄存器窗口查询到 esp 的值, 为 00AFFAC4H, 即为堆栈段的地址, 在内存窗口中查看这个地址, 如图 3.12 所示。

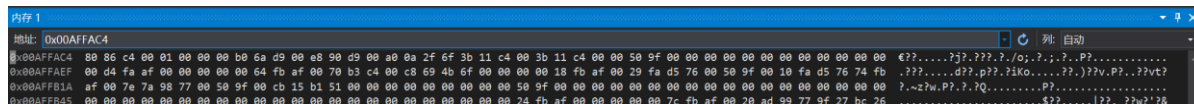


图 3.12 堆栈段地址

可以看出经过 push 操作, esp 发生了变化, 同时栈空间中的值也发生了变化。

下面随意更改几处代码, 观察编译器的提示。

将最后的END删除, 错误信息如图3.13所示。

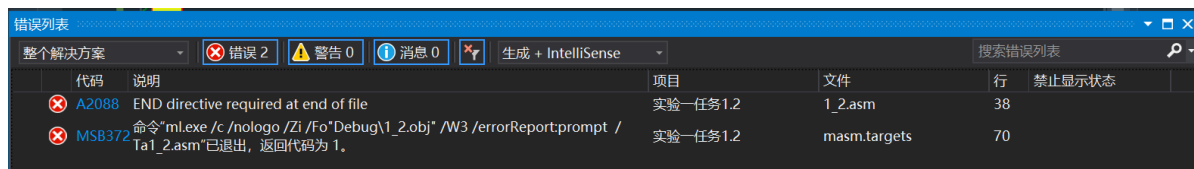


图3.13 错误信息1

3.2.2 DOSBOX 下的工具包

此处仅仅简单总结一下 DOSBOX 工具的特点:

(1) 首先, 通过使用 DOSBOX 这种通过命令行进行交互的工具, 我们可以了解编译的详细过程, 此外通过使用 dosbox 我也学习到了许多命令行的操作。

(2) 其次, DOSBOX 更为轻量级, 轻量级程序编译响应比 VS2019 要快很多并且非常准确

(3) 最后, 使用起来还是没有那么方便, 对比 vs2019, 这里调试的工具可选择性就少很多, 中断矢量表还有各类 debug 工具都显得非常简陋并且使用起来需要一定的学习成本, 而 vs2019 提供的全面与强大的图形界面使得它的学习成本较低。

3.2.3 QEMU 下 ARMv8 的工具包

此处依托于实验 5 任务 5.2, 本任务主要用 QEMU 启用 OpenEuler 操作系统, 利用 gcc 编译、gdb 调试来查看 ARMv8 体系与我们理论课上所学的 80X86 体系的异同, 重点关注了主要关注 CPU 内寄存器、段的定义方法、指令语句及格式的特点、子程序调用的参数传递与返回方法、与 C 语言混合编程、开发环境等方面。

本次任务主要通过分析测试三个示例程序来体现两种体系间的异同, 下面将依次展示实验过程中的体会。

程序 1.4.1

本程序主要显示了“Hello World!”字符串, 为单纯的汇编语言程序, 并且无子程序调用过程, 通过对该程序的测试, 可以发现 ARMv8 体系的 CPU 内寄存器、段定义方法、指令语句及格式的特点, 具体如下文所示。

汇编语言程序设计实验报告

首先利用 vi 将源程序写入到文件后，使用命令 as 将该文件编译成二进制目标文件，然后使用命令 ld 将文件链接，最终输出可执行目标文件，最终运行结果如图 3.14 所示。

```
"hello.s" [New] 15L, 186C written
[root@localhost ~]# as hello.s -o hello.o
[root@localhost ~]# ld hello.o -o hello
ld: warning: cannot find entry symbol _start; defaulting to 00000000004000b0
[root@localhost ~]# ./hello
Hello World!
```

图 3.14 编译链接运行程序 1.4.1

可以看到链接的时候给了一个警告，这应该是因为本程序中没有给出_start 来指定程序入口，所以系统自己默认给了一个入口。

然后用 gdb 来查看相关信息。

首先启用 gdb 调试该文件，用指令 r 运行，运行结果如图 3.15 所示。

```
(gdb) r
Starting program: /root/hello
Hello World!
[Inferior 1 (process 7031) exited with code 0173]
```

图 3.15 使用 gdb 调试运行程序 1.4.1

然后加断点对程序 1.4.1 进行调试，我们在 tart1 位置打一个断点，然后开始运行，通过图 3.16 可以看到程序 1.4.1 在 0x4000b4 处中断。

```
(gdb) b tart1
Breakpoint 1 at 0x4000b4
(gdb) r
Starting program: /root/hello

Breakpoint 1, 0x00000000004000b4 in tart1 ()
```

图 3.16 打断点运行中间过程图

此时，通过 info r 命令来查看 ARMv8 体系下的寄存器，部分寄存器如图 3.17 所示。

x28	0x0	0
x29	0x0	0
x30	0x0	0
sp	0xffffffffffb30	0xffffffffffb30
pc	0x4000b4	0x4000b4 <tart1+4>
cpsr	0x1000	[EL=0]
fpsr	0x0	0
fpcr	0x0	0

图 3.17 运行过程中 ARMv8 架构部分寄存器示意图

本图仅仅展示了寄存器 x0~x30 中的一部分以及部分其他寄存器。通过与 80X86 寄存器的类比以及在网上查阅资料可知，寄存器 x0~x30 应当是通用寄存器，PC 是程序计数器，类似于 80X86 中的 EIP，cpsr 和 fpsr 都是状态寄存器，而后者是前者在异常时的备份，sp 应当是栈指针。

此外，通过观察本程序代码，我们可以看出 ARMv8 下的汇编代码与 80X86 下汇编代码的指令的区别。首先对于段定义，可以看到 ARMv8 下使用 .data 段来保存数据，这就类似于 80X86 中的数据段，并且和 80X86 中的数据段定义伪指令是类似的，而 .text 段就是代码段，和 80X86 中的代码段类似。可以看到本程序中还使用了 .global 关键字，该关键字用来让一个符号对链接器可见。在指令方面，两者差别并不是很大，在 80X86 中，要获取某个变量的地址，可以直接使用 lea 命令，但是在 ARMv8 中专门给出了一个 ldr 伪指令来实现这一点，另外在使用立即数的时候，在 ARM 中可以选择使用前缀符#，但是是可以省略掉的，只是汇编器会给出一条警告。此外，ARM 采用 RISC

汇编语言程序设计实验报告

架构 CPU 和内存只能 load, store 其余全部通过寄存器, 同时支持非常多流水线操作。同时 ARM 指令集具有非常多的通用寄存器 (37 个), 但同时 x86 只有 8 个通用寄存器。Arm 指令集寻址方式不如 x86 多样。

程序 2.2.1 与 2.2.3

程序 2.2.1 和 2.2.3 在功能上都是实现一段内存的拷贝, 只是后者采用内存突发传输方式进行优化, 即一次传送 16 个字节的内存数据。

这两个程序都是使用 C 语言与汇编语言的混合编程, 因此在这里我们主要探究 ARMv8 体系架构下的子程序调用过程以及 C 与汇编语言混合编程的特点, 而对于在任务 1.4.1 处已经探究过的知识, 此处就不再赘述了。

首先可以先看一下这两个程序的运行状况与性能差别, 没有优化前的程序 2.2.1 运行结果如图 3.18 所示。优化后的程序 2.2.3 运行结果如图 3.19 所示。

```
[root@localhost ~]# gcc time.c copy.s -o m1
[root@localhost ~]# ./m1
memorycopy time is 426149088
```

图 3.18 程序 2.2.1 运行结果

```
[root@localhost ~]# gcc time.c copy21.s -o m21
[root@localhost ~]# ./m21
memorycopy time is 149346992
```

图 3.19 程序 2.2.3 运行结果

这里使用 gcc 编译 C 与汇编的混合编程程序, 可以看出优化的程度还是很高的。

接下来, 由于两个程序的差别仅仅在于 memorycopy 所使用的算法的差别, 所以我们这里仅仅对程序 2.2.1 进行探究, 来查看 ARMv8 架构中子程序调用以及 C 与汇编混合编程的特点。

首先我们可以通过 linux 下的 objdump 工具来查看可执行目标文件 m1 对应的反汇编代码, 将其重定向到 obj 文件中, 然后用 vi 查看, 搜索 memorycopy, 最终查找到 C 程序调用 memorycopy 时所采取的相关指令, 如图 3.20 所示。

```
4006e4: 90000100      adrp    x0, 420000 <clock_gettime@GLIBC_2.17>
4006e8: 91012001      add     x1, x0, #0x48
4006ec: 9001cac0      adrp    x0, 3d58000 <src+0x3937fb8>
4006f0: 911d2000      add     x0, x0, #0x748
4006f4: 9400000e      bl     40072c <memorycopy>
4006f8: 910063a0      add     x0, x29, #0x18
4006fc: aa0003e1      mov     x1, x0
400700: 52800020      mov     w0, #0x1 // #1
400704: 97ffff8b      bl     400530 <clock_gettime@plt>
```

图 3.20 调用 memorycopy 时的相关指令

从上图中, 我们可以认为在内存 0x4006e8 附近开始做调用 memorycopy 的准备, 因此我们在 gdb 中将断点打到 0x4006e8, 进行调试, 来探究 ARMv8 调用子程序时的机制。首先, 可以单步调试到调用 memorycopy 之前, 然后查看源操作数组 src 的首地址, 如图 3.21 所示, 可以看到 src 的首地址为 0x420048。随后由于反汇编代码中在调用子程序之前对多个寄存器进行了操作, 所以打印一下寄存器的值, 可以发现 x1 寄存器中的值正是 src 的首地址, 用同样的方法, 我们可以看到 x0 寄存器中存放 dst 的首地址, 而 x2 中存放子程序的第三个参数。

```
(gdb) x/x src
0x420048 <src>: 0x61616161
```

图 3.21 src 地址

通过上面的分析, 我们可以发现在本例中子程序调用时通过寄存器进行传参, 这类似于 80X86

汇编语言程序设计实验报告

的寄存器传参，即调用子程序的时候约定好采用哪一个寄存器去传递哪一个参数。此外，也可以在整个过程中查看一下其它寄存器的值，比如 `pc` 和 `sp` 这种比较常见的寄存器，可以发现 `pc` 的功能就类似于 80X86 中的 `EIP`。

而对于 C 与汇编语言混合编程，这里 ARMv8 架构和 80X86 架构应当是类似的，关键就在于模块之间的通信，对于 C 模块，这里没有什么区别，都是使用 `extern` 声明外部符号，而对于汇编模块，这里采用 `.global` 让某个符号对链接器可见，这一点和 80X86 架构基本相同，差别不大。

3.3 小结

3.3.1 VS2019

VS2019 作为 windows 环境下的经典软件，在 UI 界面、功能、扩展性等方面都达到了一个很高的水平，成为了几乎所有初学者都要学习使用的软件。对于汇编语言来说，他提供的寄存器窗口、内存窗口、反汇编窗口都比较清晰，并且图形化可视化程度高，操作起来方便，很适合初学者。界面也比较美观，字体和背景颜色都可以自己设置。

功能丰富强大也带来了笨重不轻便的特点，占用内存空间较大，程序运行时间较慢，因此更适合较大的 project 使用 VS2019，当有轻量化的编程需求时可以考虑 VSCode 这样的轻量化编程软件。

3.3.2 DosBox

DosBox 用起来不如 VS2019 方便全面，VS2019 可以提供调试的所有信息，并且有着非常全面美观的图形化，并且在熟悉 VS2019 提供的快捷键之后操作起来还是非常方便快捷的，而 DosBox 通过命令行输入，调试进入 TD 之后虽然进入了图形化界面，但是界面过于简陋，并且实际操作的时候也出现了种种让人不舒服的问题，

另外一个问题就是 DosBox 在使用的时候的窗口太小，并且经过反复的折腾也没有找到如何把 DosBox 的窗口变大，后来也是在同学的帮助下发现可以更改最初的配置文件来调大窗口，但是这一点很难自己发现，并且似乎也没有提供比较合适的帮助文档，上手还是有一定困难的。

3.3.3 QEMU 下的 ARMv8 的工具包

该工具主要在 QEMU 下运行 openEuler 操作系统，在 linux 下编译运行程序，仅提供命令行，使用 `gcc` 编译并用 `gdb` 调试，使用 `vi` 对文件进行编辑。相比于 VS2019，其学习成本是相当高的，因为不管是 `gcc`、`gdb` 还是 `vi` 都给出了一套指令系统，如果是初学者的话很容易出现种种问题。

汇 编 语 言 程 序 设 计 实 验 报 告

参考文献

- [1]许向阳. x86 汇编语言程序设计. 武汉: 华中科技大学出版社, 2020
- [2]许向阳. 80X86 汇编语言程序设计上机指南. 武汉: 华中科技大学出版社, 2007
- [3]王元珍, 曹忠升, 韩宗芬. 80X86 汇编语言程序设计. 武汉: 华中科技大学出版社, 2005
- [4]汇编语言课程组. 《汇编语言程序设计实践》任务书与指南, 2022